Available online at https://ejournal.corespub.com/index.php/ijmsc/index

International Journal of Mathematics, Statistics, and Computing

e-ISSN 3025-0803

Vol. 2, No. 2, pp. 85-89, 2024

Optimization of White Box Testing by Utilizing Branching and Repeating Structures in Java Programs Using Base Path

Kalfin[1*], Riza Andrian Ibrahim[2], Grida Saktian Laksito[3]

[1]*Statistics Study Program, Faculty of Science, Technology and Mathematics, Matana University*
[2]*Doctoral of Mathematics Study Program, Universitas Padjadjaran, Jatinangor, West Java, Indonesia*
[3] *Faculty of Business, Economics, and Sosial Development, Terengganu, Malaysia*

**Corresponding author email: kalvin@matanauniversity.ac.id*

Abstract

Software testing is a process created to detect anomalies in the operation of a program or system in order to achieve the expected results. White box testing is a software testing method that tests the internal structure, design and program code. This research aims to produce an optimization method for white box testing in Java programs by utilizing branching and repetition structures using the basis path method, as well as analyzing the effectiveness of the proposed method in generating test cases. The Java program tested in this research includes input function calls, loops, branching, and exception handling. Test case design is carried out by applying the basis path method to achieve comprehensive coverage. The test results show that the base path method is able to produce effective test cases for testing control structures without redundancy. Test case design is assisted by flowgraph and matrix graph modeling. It is hoped that this research can contribute to the optimization of white box testing techniques.

Keywords: Software testing, white box testing, Java programs, basis path, matrix graph.

1. Introduction

Testing, or software testing, is a process created to detect anomalies in the operation of a program or system to achieve expected results. Testing uses software to carry out various methods under controlled conditions, such as checking whether the system being built meets specifications, then detecting errors or looking for errors in the system, and determining whether the system is ready for use by users. or not. Apart from that, testing is also useful for evaluating the work performance of developers or programmers (Henard et al., 2016; Mishra et al., 2020; Arora et al., 2016).

Testing plays an important role in software development, so good test planning must be considered. In software testing theory, there are several software testing methods, one of which is white box testing. In this research, the testing carried out was White Box Testing, because usually many developers directly carry out usability and functionality testing, for example by testing the system user interface, or what is usually called black box testing. The testing method using white box testing is still rarely carried out, while the white box testing method is the initial stage of software testing before testing the system appearance, so that the program structure and logic flow work according to business operations (Habibi and Mirian-Hosseinabadi 2015; Anbunathan and Basu 2018).

The processes running on the system and the source code of the software created must be compatible with the system so that it is suitable for use by potential users of the system. White box testing is testing based on inspection of design details, where the software design control structure is used procedurally to divide testing into several test cases. In other words, white box testing is a programming guide to get images that are 100% correct. The white box testing method emphasizes testing the program source code (Honest, 2019; Skalka and Drlík 2023). In white box testing, testing is carried out by looking at the program code in the module and analyzing whether there are errors or not.

If the module produces output that is not in accordance with the business process being carried out, the program code is checked and then corrected so that the output produced is as expected. The white box test method is very effective in determining design plans, decisions, opinions, finding bugs in programs and finding implementation bugs in software. The white box testing method has several testing techniques including extract coverage, branch coverage, multicondition coverage, loop coverage, call coverage, and path coverage (Nordeen, 2020; Takeda et al., 2019).

The aim of this research is to produce an optimization method for White Box testing of Java programs using a branching and iterative structure using the basic path method and to analyze the effectiveness of the proposed method in generating test cases. It is hoped that this research can contribute to the development of knowledge in the field of software testing, especially those related to optimizing white box testing. In addition, this research is expected to bring practical benefits to developers in controlling the Java programs they develop.

2. Methodology

2.1. From the whiteboxtesting.java program that will be tested

```
   try {
      Scanner scanner = new Scanner (System.in);
      System.out.print("Enter number: ");
      int number = scanner.nextInt();
      for (int i = 1; i <= 5; i++) {
         String statusNumber = checkNumber(number);
         System.out.println("Iteration" + i + ": Number " + statusNumber + ".");
         printNumberAndSquare(number);                                              (1)
         number += 1;
      }
      scanner.close();
      System.out.println("Program completed.");
      } catch (Exception e) {
      System.out.println("An error occurred: " + e.getMessage());
}
```

The program code begins with a try-catch block to handle exceptions, then a Scanner declaration to receive input from the user, then the program asks the user to enter a number, which is stored in an integer type number variable. After that there is a for loop of 5 iterations. In the loop, the checkNumbers() method is called which accepts a number parameter. This method returns the status of the number, whether positive, negative or zero, which is stored in the variable statusNumber. Then print the number of iterations and the status number. Next, call the printNumberAndSquare() method which prints the number and its square. After that, the number value is increased by 1 for the next iteration. After the loop is complete, the Scanner is closed and a message is printed that the program has finished. The catch block handles exceptions by displaying the error message that occurred. Overall, this program code will be tested in a white box manner to ensure that the logic and implementation are correct, especially the use of control structures such as loops, branching and exception handling.

2.2. Simple program algorithm in the Java programming language

The flowchart in this research is a representation of a simple program in the Java programming language. It displays a welcome message, requests user input for a number, and then calculates and displays the square of that number in a loop for five iterations (Horváth et al., 2019).

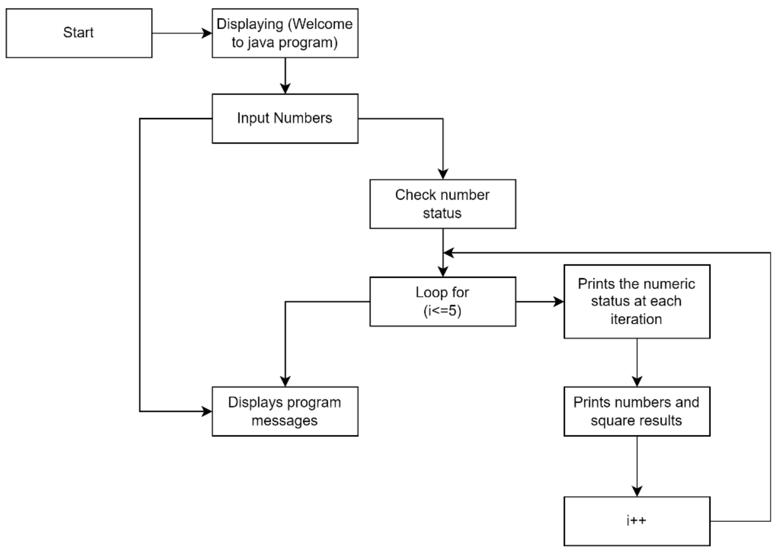
Start
Displaying (Welcome to java program)
Input Numbers
Check number status
Loop for (i<=5)
Prints the numeric status at each iteration
Prints numbers and square results
i++
Displays program messages

Figure 1: Flowchart

a). At the top of the flowchart, there is a command to display the text "Welcome to the Java Program". It will display a welcome message when the program starts.
b). After that, the program will ask for input numbers from the user. This is done with the "Number Input" command.
c). After getting the input number, the program will display the message "Program Completed". This means that the program will stop after performing the next steps.
d). Before the program stops, it will check the status of the numbers entered by the user. This is done using a "Loop for" loop, which will repeat as long as the variable "i" is less than or equal to 5.
e). At each iteration of the loop, the program will display the status of the numbers using the command "Print Status of Numbers at Each Iteration".
f). Apart from that, the program will also display the value of the number and its squared result using the "Print Number and Squared Result" command.
g). Once the loop is complete, the program will return to step 3 and display the message "Program Completed".

3. Results and Discussion

3.1. Flowgraph

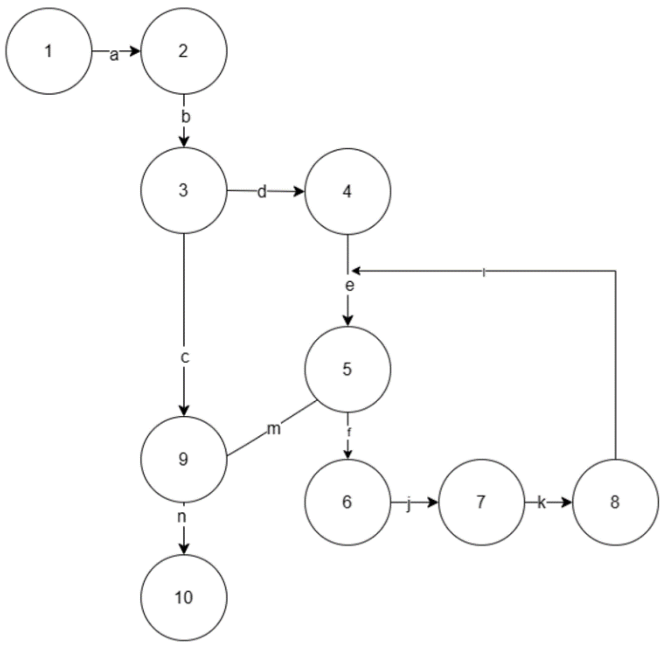

Figure 2: Flowgraph

R1: a-b-c-n
R2: a-b-d-e-m-n (2)
R3: a-b-d-e-j-k-l-m-n

Based on the flow graph in Figure 2, it is known that edge (E) = 11 which is a line connecting nodes, the number of nodes (N) = 10 which is a circle that describes an activity, the number of predicates (P) = 2 which are branching nodes, and the number of regions (R) = 3 which indicates an area in the flow graph, which can be seen with the symbols R1 to R4.

3.2. Cyclomatic Complexity

Cyclomatic Complexity calculations are carried out to determine the complexity of the control flow of a program. In the program code analyzed, Cyclomatic Complexity calculations were carried out using three approaches, namely based on the number of edges and nodes in the flowgraph, the number of predicate nodes, and the number of regions. Cyclomatic Complexity calculations are very important to do at an early stage before designing test cases for white box testing. This value determines the basis set of independent paths that must be tested in the program for testing to achieve comprehensive coverage (Squire et al., 2020; Yang, 2013; Chen et al., 2014).

$$\begin{array}{lll} V(G)\ E - N + 2 & V(G) = P + 1 & V(G) = R \\ V(G) = 11 - 10 + 2 & V(G) = 2 + 1 & V(G) = 3 \\ V(G) = 3 & V(G) = 3 & \end{array} \qquad (3)$$

From these three approaches, the same Cyclomatic Complexity value is obtained, namely V(G) = 3. This indicates that the program has three independent paths that need to be tested. This value is obtained because the program only has 2 predicate nodes/branching nodes and forms 3 regions in the flowgraph. The number of edges is 11 and nodes are 10. The low value of Cyclomatic Complexity in this program shows that the program is relatively simple with few logical branches/branches. The higher the Cyclomatic Complexity value, the more complicated the program is.

3.3. Independent Path and Graph Matrix

Based on the results of the Cyclomatic Complexity calculation above, the number of independent path results for the whiteboxtesting.java program code is 3 with the following independent path and graph matrix:

Path 1 = a-b-c-n
Path 2 = a-b-d-e-m-n (4)
Path 3 = a-b-d-e-f-j-k-l-m-n

Graph matrices are used in white box testing techniques to represent the relationships between nodes in the flow graph of the program being tested. The graph matrix is in the form of a two-dimensional table with rows and columns that represent nodes in the flow graph. The main purpose of the graph matrix is to determine the test path that must be carried out to achieve comprehensive test coverage. With a graph matrix, testers can identify which nodes belong to a particular execution path (Arwan and Rusdianto 2017; Shuaibu et al., 2019; Padmanabhan, 2022). Apart from that, the graph matrix is also useful for tracking which nodes have been covered and which have not been covered by the test cases that have been designed. In this way, we can avoid missing paths when designing test cases.

Table 1: Graph matrix

	1	2	3	4	5	6	7	8	9	10
1		a								
2			b							
3				d					c	
4					e					
5						f			m	
6							j			
7								k		
8					l					
9										n
10										

Path 1 is a normal path without branches. Line 2 passes through branch "d" and line 3 passes through branches "d" and "f". Based on the matrix, it can be seen that the 3 independent paths have been covered by the test cases carried out. All nodes have been traversed on one of the test paths. Thus, it can be concluded that white box testing in this program has achieved comprehensive coverage based on its independent path. Matrix graphics have helped ensure no paths are missed during testing.

4. Conclusion

White box testing by utilizing the branching and repetition structure in Java programs using the basis path method is an effective and efficient testing method in ensuring software quality. This method allows designing test cases that can cover all execution paths in the program, thereby ensuring comprehensive test coverage. White box testing is also important in educational contexts, helping students understand the logic and flow of programs.

References

Anbunathan, R., & Basu, A. (2018). Basis Path Based Test Suite Minimization Using Genetic Algorithm. International Journal of Intelligent Systems and Applications, 10(11), 36.

Arora, V., Bhatia, R., & Singh, M. (2016). A systematic review of approaches for testing concurrent programs. Concurrency and Computation: Practice and Experience, 28(5), 1572-1611.

Arwan, A., & Rusdianto, D. S. (2017). Optimization of Genetic Algorithm Performance Using Naïve Bayes for Basis Path Generation. Kinetik: Game Technology, Information System, Computer Network, Computing, Electronics, and Control, 273-282.

Chen, J., Peng, H., & Xie, X. (2014). Improvement of the Baseline Method in Structural Testing. The Open Automation and Control Systems Journal, 6(1).

Habibi, E., & Mirian-Hosseinabadi, S. H. (2015). Event-driven web application testing based on model-based mutation testing. Information and Software Technology, 67, 159-179.

Henard, C., Papadakis, M., Harman, M., Jia, Y., & Le Traon, Y. (2016, May). Comparing white-box and black-box test prioritization. In Proceedings of the 38th International Conference on Software Engineering (pp. 523-534).

Honest, N. (2019). Role of testing in software development life cycle. International Journal of Computer Sciences and Engineering, 7(5), 886-889.

Horváth, F., Gergely, T., Beszédes, Á., Tengeri, D., Balogh, G., & Gyimóthy, T. (2019). Code coverage differences of Java bytecode and source code instrumentation tools. Software Quality Journal, 27, 79-123.

Mishra, D. B., Acharya, A. A., & Acharya, S. (2020). White box testing using genetic algorithm—An extensive study. A journey towards bio-inspired techniques in software engineering, 167-187.

Nordeen, A. (2020). Learn Software Testing in 24 Hours: Definitive Guide to Learn Software Testing for Beginners. Guru99.

Padmanabhan, M. (2022, November). Regression Test Case Optimization Using Jaccard Similarity Mapping of Control Flow Graph. In International Conference on Innovations in Computational Intelligence and Computer Vision (pp. 545-558). Singapore: Springer Nature Singapore.

Shuaibu, I., Musa, M., & Ibrahim, M. (2019). Investigation onto the software testing techniques and tools: An evaluation and comparative analysis. International Journal of Computer Applications, 177(23), 24-30.

Skalka, J., & Drlík, M. (2023). Automatic Source Code Evaluation Test Development in Programming Education Using Grey-box Methods. IEEE Access.

Squire, M. D., Maynard-Nelson, L. A., Brown, T. A., Crumbley, R. T., Holzmann, G. J., Jennings, M., ... & Marchetti, J. D. (2020). Cyclomatic Complexity and Basis Path Testing Study (No. NESC-RP-20-01515).

Srivatsava, P. R., Mallikarjun, B., & Yang, X. S. (2013). Optimal test sequence generation using firefly algorithm. Swarm and Evolutionary Computation, 8, 44-53.

Takeda, T., Takahashi, M., Yumoto, T., Masuda, S., Matsuodani, T., & Tsuda, K. (2019, April). Applying change impact analysis test to migration test case extraction based on idau and graph analysis techniques. In 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) (pp. 131-139). IEEE.